Homework 5

1

a.)

Without knowing the language, the language is clearly dynamically typed since the variable user_id can be assigned a different type throughout the lifetime of the variable. In the beginning, it was assigned a string and in the end it was assigned as an int value.

b.)

This language uses lexical scoping strategy because when the function boop() is called, it is only printing the x variable that's in the respective context of the function. So for boop() only the var x = 2; is in scope. Although we call beep(), once we jump out of the beep function, the x =1; assignment of the variable x is no longer in scope, thus the only x that is used in console.log(x) at the end of the boop() function is console.log(x); This is similar to other lexical scoping languages, like C++.

c.)

This programming language is also lexically scoped. This is because when the print statement in the inner let function on lines 3-6 are executed, we only consider the scope of the lines within those {..}. However, once we jump out of it, the x = 0 is then printed out again. Additionally, when we declare a new variable, we can see that the type is assigned and only considers the scope of the most recent change. That's why it then prints the message.

2

a.)

The scope of name is that it is in scope for the duration of the function boop(). The lifetime of name is also within the duration of the function boop(). The same scoping and lifetime will occur for the variable res but the object bound to res will have a different lifetime. The object bound to res since it is returned from the function will have a lifetime outside of the duration of the function boop().

b.)

The following C++ coe is considered undefined behavior because the assignment of the n variable is technically out of the scope for n. So when the address is assigned to the variable n, technically in a lexically scoped language, the variable shouldn't be able to be assigned within the curly braces.

c.)

The code does still work likely because of shadowing which is one of the tricks that C++ uses when it has variables of the same name but within different scopes.

3

a.)

We likely want the type of refCount to be a pointer to an int. That way we can have the shared pointers be able to access the refCount. If there was no pointer to an int and just an int, there would be no way for the other shared pointers to have access to the refCount in a concurrent manner.

b.)

```
my_shared_ptr(int * ptr) {
    this->ptr = ptr;
    refCount = new int(1);
}
```

c.)

```
my_shared_ptr(const my_shared_ptr & other) {
    ptr = other.ptr;
    refCount = other.refCount(1);
    (*refCount)++ // need to dereference this pointer before you could ++
}
```

d.)

```
~my_shared_ptr()
{
    (*refCount)-- // ideally it will return to you to be destroyed when refCount=1
    if (*refCount == 0) {
        if (nullptr != ptr) {
            delete ptr;
        }
        delete refCount;
    }
}
```

e.)

```
my_shared_ptr& operator=(const my_shared_ptr & obj)
{
    if (this == &other) {
        return *this;
    }

    // Decrement the reference count for the current object
    if (refCount && --(*refCount) == 0) {
        delete ptr;
        delete refCount;
    }

    // Share the resource
    ptr = other.ptr;
```

```
    refCount = other.refCount;

    // Increment the reference count for the new object
    if (refCount) {
        (*refCount)++;
    }

    return *this;
}
```

4.

a.

The key with using something like Rust that has better behavior for this type of task is because that way the behavior of the program is more predictable. Garbage collection is essentially a more handsoff when it comes to program development and so if we want full control of the program its best to utilize a language that doesn't use garbage collection.

b.

I disagree, reference counting is at its core a type of garbage collection and thus the unpredictable and non-deterministic behavior exhibited by using reference counting would be undesirable for our space program.

c.

Some advice that I would give to Kevin is that he should decide based on what his specific project requirements are. Go's philosophy of garbage collections yields a more predictable and low-latency performance. There is a more hands-on configuration of the memory management model used in Go. However, if Kevin wants a more higher level abstraction from the memory management model, C# would likely be a more higher level model. Thus, the development experience for Kevin might be a fair bit easier with C#.

d.

Go and C++ have very different ways of managing their destructors and in go's case finalizers. Destructors are executed on objects at the end of their lifetime, meanwhile the go finalizers are executed during garbage collection. Thus the sockets may still be active since garbage collection occurs later on in the program's running process.

5.

For the lines involving const int, unsigned int, and short, a cast is performed. The instruction mov1 basically "moves" the bits by copying them into a different location. Meanwhile a complete conversion is occurring for the booleans and floats. cvtsi2ssl is used to make these conversions in order to create code to actual perform the conversion.